ORCHESTRER LE CODE

Architecture, Git, tests, agents et mise en production expliqués à ceux qui décident

La promesse
Comprendre assez profondément le logiciel pour donner une direction claire à un agent de code, vérifier son travail et publier sans jouer à la roulette russe.

MANUEL DE CHEVET • ÉDITION 2026

Une méthode de vibe coding intelligent

Avant-propos

Ce livre n’essaie pas de faire de toi un développeur en accéléré. Il vise une compétence différente : devenir le maître d’ouvrage lucide d’un système logiciel. Tu dois savoir poser le problème, découper le travail, reconnaître une décision structurante, demander une preuve et refuser une mise en production mal préparée.

Un agent sait produire beaucoup de code. Il ne connaît pourtant ni ton entreprise, ni le coût futur d’une dépendance, ni la gravité réelle d’une perte de données. Sa vitesse amplifie la qualité de ta méthode - ou l’absence de méthode. Le vrai levier n’est donc pas le prompt magique. C’est le harnais : contexte, règles, étapes, tests, revues, traces et limites.

> Idée centrale
> Le code est un matériau. L’architecture organise ce matériau. Git garde son histoire. Les tests apportent des preuves. Le déploiement transforme une version vérifiée en service réel.

Sommaire

CHAPITRE 1

Voir le logiciel comme un système

Avant d’écrire une ligne, il faut comprendre ce qui circule, ce qui décide, ce qui persiste et ce qui peut casser.

Une application est une organisation

Imagine un atelier. L’interface est le comptoir. La logique métier est le savoir-faire. La base de données est l’archive. L’API est le bordereau qui fait circuler les demandes. Le serveur est le bâtiment équipé. Le réseau est la route. Une application fiable attribue clairement chaque responsabilité.

Les quatre mouvements fondamentaux

1. Recevoir
   Une intention entre : clic, formulaire, fichier, appel API.
2. Valider
   Le système vérifie format, droit, cohérence et règles métier.
3. Transformer
   Il calcule, génère, classe ou déclenche une action.
4. Persister ou répondre
   Il enregistre un état durable et retourne un résultat observable.

État, données et comportement

L’état est la situation actuelle du système : utilisateur connecté, commande payée, fichier traité. Les données sont la représentation durable de cet état. Le comportement est la règle qui fait passer d’un état à un autre. Beaucoup de bugs viennent d’une transition autorisée au mauvais moment ou répétée deux fois.

> Question d’architecte
> Quelles données entrent ? Qui peut les modifier ? Quelle règle s’applique ? Quel résultat doit être observable ? Que se passe-t-il si l’opération est répétée ou interrompue ?

Les frontières à rendre visibles

Une frontière sépare deux responsabilités : navigateur et serveur, application et base, service et fournisseur externe. À chaque frontière, exige un contrat : données attendues, réponses possibles, erreurs, délai, sécurité. Une interface floue déplace les bugs au lieu de les résoudre.

Couplage et cohésion

La cohésion mesure si un module fait une chose qui a du sens. Le couplage mesure combien il dépend des autres. Vise une forte cohésion et un couplage faible : le module Facturation connaît les factures, mais ne devrait pas décider du dessin des boutons ni connaître les détails du serveur de courriel.

Du texte source au comportement réel

Le code source est une description. Un compilateur ou un interpréteur la transforme en instructions exécutables. Au démarrage, le système d’exploitation crée un processus, lui attribue mémoire, fichiers, réseau et temps processeur. Comprendre cette chaîne aide à distinguer un défaut de code, un défaut de configuration et un défaut d’environnement.

Pile, tas et durée de vie

La pile contient surtout le contexte des appels de fonctions et disparaît naturellement au retour. Le tas contient les objets dont la durée de vie est plus flexible ; un ramasse-miettes ou le programme doit libérer ce qui n’est plus utile. Une fuite mémoire signifie qu’une référence maintient vivant un objet inutile. Une saturation n’est donc pas forcément un manque de RAM : elle peut révéler une durée de vie mal maîtrisée.

Entrées, sorties et effets de bord

Une fonction pure transforme des entrées en sortie sans modifier le monde extérieur. Un effet de bord écrit en base, envoie un message, lit l’heure, produit un fichier ou appelle un service. Les effets sont nécessaires, mais les isoler rend les règles testables et les échecs contrôlables.

> Lecture d’ingénieur
> Demande toujours : cette fonction calcule-t-elle ou agit-elle ? Si elle agit, l’action peut-elle échouer, être répétée, compensée ou observée ?

Synchronisme, asynchronisme et concurrence

Une opération synchrone bloque son scénario jusqu’au résultat. Une opération asynchrone permet d’attendre sans immobiliser tout le système. La concurrence signifie que plusieurs travaux progressent durant la même période ; le parallélisme qu’ils utilisent réellement plusieurs unités de calcul. Ces notions ne rendent pas automatiquement un programme plus rapide : elles introduisent ordre incertain, annulation, délais et courses critiques.

Une course critique apparaît lorsque le résultat dépend de l’ordre d’opérations concurrentes. Exemple : deux requêtes lisent « stock = 1 », puis vendent chacune l’unité. La correction se place souvent dans la base avec transaction, verrou ou contrainte, pas seulement dans l’interface.

CHAPITRE 2

Architecture : séparer pour maîtriser

Une bonne architecture n’est pas spectaculaire. Elle rend le changement local, la panne compréhensible et la décision réversible.

Les couches utiles

1. Présentation
   Ce que l’utilisateur voit et manipule.
2. Application
   Les scénarios : créer une commande, valider un devis, exporter un PDF.
3. Domaine
   Les règles qui donnent son sens au produit.
4. Infrastructure
   Base, fichiers, courriels, paiements, services externes.

Cette séparation n’impose pas quatre serveurs ni un framework complexe. Elle peut exister dans un monolithe simple, sous forme de dossiers et de dépendances orientées proprement.

Monolithe modulaire d’abord

Pour la plupart des projets métier, commence par une seule application déployable, découpée en modules cohérents. Les microservices ajoutent réseau, authentification interservices, observation distribuée et déploiements multiples. Ils résolvent surtout des problèmes d’échelle organisationnelle déjà prouvés.

Décision par défaut
Une base de code, un déploiement, des modules clairs. Extraire un service seulement lorsqu’une contrainte mesurée le justifie : charge indépendante, sécurité particulière ou équipe autonome.

API, contrat et idempotence

Une API est une frontière programmable. Son contrat doit préciser la requête, la réponse, les codes d’erreur et les droits. Une opération idempotente produit le même résultat si elle est répétée : indispensable pour les paiements, webhooks et reprises après coupure.

Dépendances et dette

Une dépendance économise du travail mais crée une obligation : mises à jour, vulnérabilités, changement de licence, disparition du projet. Demande à l’agent pourquoi elle est nécessaire, quelle alternative standard existe et comment la remplacer. La dette technique n’est pas du mauvais code en général ; c’est un raccourci dont le coût futur est connu ou ignoré.

Les qualités non fonctionnelles

Une fonctionnalité dit ce que fait le système. Les qualités non fonctionnelles disent comment il doit le faire : sécurité, performance, disponibilité, accessibilité, maintenabilité, souveraineté des données. Elles doivent devenir des critères mesurables, pas des adjectifs.

Ports, adaptateurs et inversion des dépendances

Le domaine peut définir un port : « enregistrer une commande » ou « envoyer une notification ». L'infrastructure fournit un adaptateur PostgreSQL, SMTP ou autre. Le cœur dépend ainsi d'une capacité abstraite et non du fournisseur concret. On peut tester avec un faux adaptateur et remplacer la technologie sans réécrire la règle métier.

Transactions et cohérence

Une transaction regroupe des opérations qui doivent réussir ensemble. ACID résume quatre propriétés : atomicité, cohérence, isolation et durabilité. Mais une transaction locale ne couvre pas naturellement un paiement externe et un courriel. Dans un flux distribué, on utilise souvent une boîte d'envoi transactionnelle, des événements idempotents et des compensations plutôt qu'une transaction magique sur tout le monde.

Cohérence forte ou éventuelle

Avec une cohérence forte, toute lecture voit immédiatement la dernière écriture confirmée. Avec une cohérence éventuelle, plusieurs composants convergent après un délai. Un tableau de bord peut tolérer quelques secondes ; l'unicité d'un paiement beaucoup moins. Le métier choisit ce qui peut être différé.

Architecture évolutive, pas prédictive

L'objectif n'est pas de deviner cinq années d'évolution. Il est de garder des frontières propres, des contrats testés et des migrations possibles. Une décision réversible peut être prise vite. Une décision difficile à renverser - modèle de données, fournisseur captif, identité, découpage réseau - mérite davantage de preuve et un ADR.

CHAPITRE 3

Concevoir avant de coder

Le meilleur moyen d’aller vite consiste à réduire l’ambiguïté avant qu’elle ne se transforme en fichiers, tables et dépendances.

Du problème à la tranche verticale

Décris d’abord le résultat humain : « un client dépose un fichier et reçoit une confirmation ». Une tranche verticale traverse juste assez d’interface, logique et stockage pour produire ce résultat. Elle est démontrable et testable. Évite les grands chantiers horizontaux du type « faire toute la base », puis « faire toute l’interface ».

La fiche de fonctionnalité

1. Intention
   Pour qui ? Quel problème ? Quelle amélioration observable ?
2. Scénario nominal
   Le chemin simple, étape par étape.
3. Règles
   Ce qui est autorisé, interdit, calculé ou conservé.
4. Cas limites
   Données absentes, doublons, coupure, lenteur, droits insuffisants.
5. Critères d’acceptation
   Des phrases vérifiables qui définissent terminé.
6. Hors périmètre
   Ce que cette tranche ne tente pas de résoudre.

ADR : enregistrer les décisions

Un Architecture Decision Record tient souvent sur une page : contexte, décision, options rejetées, conséquences. Il évite que l’agent ou un futur collaborateur redécouvre l’histoire en lisant le code. Exemples : choix de PostgreSQL, stockage S3, monolithe modulaire, stratégie d’authentification.

> Bon critère d’acceptation
> « Après un double clic sur Envoyer, une seule commande est créée et le bouton affiche l’état en cours » est vérifiable. « Le formulaire fonctionne bien » ne l’est pas.

Modèle de données : nommer le réel

Les tables et objets sont un vocabulaire du métier. Choisis des noms stables et non ceux de l'écran du moment. Décide les identifiants, relations, contraintes d'unicité et règles de suppression. Toute modification de structure doit passer par une migration versionnée.

Invariants : les lois du produit

Un invariant doit toujours rester vrai : le total d'une facture égale ses lignes, une commande payée possède une référence de paiement unique, un utilisateur ne lit que son organisation. Les invariants importants doivent être défendus au niveau le plus fiable : règle métier, contrainte de base et test. L'interface seule n'est jamais une frontière de sécurité.

Contrats et types

Un type réduit l'espace des états possibles. « Chaîne de caractères » est faible ; « adresse e-mail validée » ou « montant positif en centimes » exprime une règle. Aux frontières, valide les données reçues et produis des erreurs structurées. À l'intérieur, manipule des objets déjà valides. Cette discipline simplifie le raisonnement de l'agent comme celui de l'humain.

Lire le risque avant la solution

1. Réversibilité
   Peut-on retirer ce choix sans migrer tout le produit ?
2. Rayon d’explosion
   Combien d’utilisateurs, données ou services seraient touchés ?
3. Détectabilité
   Saurons-nous rapidement qu’il échoue ?
4. Récupération
   Existe-t-il une compensation, restauration ou reprise ?
5. Connaissance
   La décision est-elle comprise et documentée, ou seulement générée ?

Le diagramme minimal

Pour une fonctionnalité, un diagramme suffit souvent : acteur, interface, service applicatif, base et éventuel fournisseur. Trace les flèches dans l’ordre et annote les écritures, délais et erreurs. Le dessin n’a de valeur que s’il permet de poser une question impossible à voir dans une liste de fichiers.

CHAPITRE 4

Le harnais agentique

Le harnais est le système qui transforme un modèle génératif rapide en collaborateur contraint, informé et vérifiable.

Ce que contient un bon harnais

1. Carte du dépôt
   Où sont l’interface, le domaine, les tests, les scripts et la documentation.
2. Règles permanentes
   Conventions, commandes autorisées, sécurité, responsabilités des modules.
3. Contexte de tâche
   Problème, preuves, fichiers concernés, critères d’acceptation.
4. Boucle de preuve
   Tests, analyse statique, revue du diff, exécution réelle.
5. Garde-fous
   Pas de secret dans Git, pas de destruction, pas de déploiement implicite.
6. Mémoire durable
   ADR, documentation, journal de versions et procédures.

Demander une enquête avant une modification

Une bonne mission commence par inspecter. Demande : reproduis le problème, identifie la cause, propose le changement minimal, liste les risques et attends ou exécute selon l'autorisation donnée. Une erreur courante est de confondre le symptôme visible avec la cause réelle.

Un prompt opératoire

```
Contexte : ce projet fait…
Objectif : obtenir…
Contraintes : préserver…, ne pas…
Acceptation : étant donné…, quand…, alors…
Méthode : inspecte, propose, modifie par petites étapes, teste, relis le
diff.
Livraison : résume les fichiers, preuves, risques restants et procédure de
retour arrière.
```

Les rôles utiles

Le Pilote découpe et arbitre. Le Bâtisseur implémente. Le Testeur cherche les échecs. Le Critique challenge la conception. Le Gardien vérifie sécurité et règles. L'Éclaireur recherche les options. L'Archiviste maintient les traces. L'Analyste mesure. Ces rôles peuvent être joués successivement par un agent ; l'indépendance de la revue compte davantage que le nombre d'agents.

> Règle d'or
> Ne demande jamais seulement « fais la fonctionnalité ». Demande aussi quelles preuves montreront qu'elle fonctionne et quelles conséquences elle crée.

Le protocole de conversation en sept messages

1. Mission
   Tu donnes résultat, contexte, contraintes et niveau d’autorisation.
2. Reformulation
   L’agent reformule le problème, les inconnues et les critères.
3. Inspection
   Il lit règles, architecture, état Git, code et tests concernés.
4. Proposition
   Il expose cause ou conception, fichiers, risques et plan minimal.
5. Autorisation
   Tu arbitres les choix structurants et fixes le périmètre.
6. Exécution
   Il modifie, teste et rend compte à intervalles utiles.
7. Handoff
   Il livre preuves, diff, limites, déploiement et retour arrière.

Niveaux d’autorisation

Sépare clairement les verbes. « Analyse » autorise la lecture et le diagnostic, pas la modification. « Implémente » autorise les changements locaux et tests sûrs. « Prépare une PR » autorise commit et push si le dépôt le permet. « Déploie » est une autorisation distincte, plus risquée. Un bon harnais encode ces limites au lieu de les réinventer dans chaque conversation.

Le contexte en couches

Le contexte stable décrit le produit, l'architecture et les règles du dépôt. Le contexte de domaine explique le vocabulaire et les invariants. Le contexte de tâche décrit le résultat du jour. Les preuves dynamiques viennent des commandes, logs, tests et diff. Mettre tout dans un prompt géant dilue l'essentiel ; donne une carte puis laisse l'agent charger les détails nécessaires.

Quand interrompre l'agent

Arrête la boucle lorsqu'il rencontre une ambiguïté métier, une donnée sensible inattendue, une migration destructive, un secret absent, un dépôt sale qui chevauche le travail, un test incohérent ou une action externe non autorisée. Une difficulté technique ordinaire doit au contraire être poursuivie dans le périmètre convenu.

Anti-patterns de dialogue

« Fais au mieux » sans critères ; demander plusieurs refontes dans la même tâche ; imposer une bibliothèque sans expliquer la contrainte ; accepter « les tests passent » sans commande ni résultat ; laisser l'agent corriger silencieusement des tests pour faire du vert ; confondre un aperçu local avec une version publiée.

CHAPITRE 5

L’ordre juste pour construire

Coder simplement, c’est obtenir tôt une preuve complète, puis élargir sans perdre la maîtrise.

1	Observer Lire le dépôt, exécuter l’existant, reproduire le besoin ou le bug.
2	Définir Écrire scénario, critères, limites et risque principal.
3	Dessiner Choisir le flux, les responsabilités et le modèle de données minimal.
4	Préparer Créer une branche, vérifier l’état Git, identifier les commandes de test.
5	Prouver Écrire un test ou une reproduction qui échoue pour la bonne raison.
6	Implémenter Faire le plus petit changement cohérent.
7	Vérifier Tests ciblés, suite complète, analyse statique et essai humain.
8	Relire Inspecter le diff : intention, bruit, secrets, migrations, dépendances.

9	Documenter Mettre à jour ADR, README, configuration et journal si nécessaire.
10	Publier Commit clair, push, PR, CI, merge, déploiement progressif, observation.

La boucle rouge, verte, propre

Rouge : une preuve échoue et démontre le manque. Vert : le changement minimal la fait passer. Propre : on simplifie sans changer le comportement. Même sans test-first strict, cette logique évite d’écrire un test qui ne fait que confirmer son propre code.

Petit ne veut pas dire morcelé

Un changement doit être petit mais complet : une intention, une preuve, une conséquence lisible. Dix micro-commits incompréhensibles ne sont pas supérieurs à un commit cohérent. La taille se juge par la capacité à relire et revenir en arrière.

Definition of Done

> Terminé signifie
> Critères satisfaits ; tests pertinents réussis ; erreurs traitées ; logs utiles ; sécurité examinée ; documentation à jour ; migration et retour arrière prévus ; diff relu ; déploiement observable.

Séquence détaillée d’une session agentique

1. Établir le point de départ
   Branche, commit, modifications existantes, commandes connues.
2. Faire produire une carte
   Flux actuel, symboles, tests, frontières et inconnues.
3. Choisir une tranche
   Un résultat démontrable, pas une couche technique complète.
4. Créer la preuve initiale
   Test qui échoue ou protocole de reproduction enregistré.
5. Modifier près de la cause
   Éviter les refontes voisines non nécessaires.
6. Faire une vérification croissante
   Test ciblé, module, intégration, parcours réel.
7. Inspecter les conséquences
   Diff, dépendances, schéma, configuration, sécurité.
8. Obtenir une revue hostile
   Chercher comment casser la solution et quels tests manquent.
9. Préparer la livraison
   Commit, notes, migration, monitoring, rollback.
10. Observer après publication
    Comparer métriques et comportement aux attentes.

Une tâche, plusieurs boucles

La macro-boucle va du besoin à la production. À l’intérieur, chaque hypothèse suit une micro-boucle : observer, prédire, modifier, mesurer. Si l’agent change cinq variables avant de vérifier, il perd la capacité d’attribuer le résultat à une cause. Cette discipline scientifique est particulièrement importante en débogage et performance.

CHAPITRE 6

Git sans folklore

Git est une machine à enregistrer des états et à organiser la collaboration. Il ne remplace ni la sauvegarde des données ni le déploiement.

Le modèle mental

Le dossier de travail contient tes modifications. La zone de préparation sélectionne ce qui entrera dans le prochain commit. Le commit est un instantané nommé avec un parent. La branche est une étiquette mobile pointant vers une suite de commits. Le dépôt distant permet le partage et la sauvegarde de l’histoire du code.

Les verbes essentiels

Commande	Sens
status	Voir la situation avant toute action.
diff	Lire les changements non préparés.
add	Choisir ce qui entrera dans le commit.
commit	Enregistrer un état cohérent localement.
push	Envoyer les commits vers le dépôt distant.
pull	Récupérer et intégrer des changements distants.
fetch	Récupérer sans intégrer.
merge	Réunir deux histoires.
revert	Créer un commit qui annule un ancien changement.

Qu’est-ce qu’une PR ?

Une Pull Request - PR - est une proposition d’intégrer une branche dans une autre. Elle montre le diff, porte la discussion, déclenche la CI et conserve l’accord. Utilise-la pour tout changement de production qui mérite une seconde lecture, touche aux données, à la sécurité, à l’architecture ou dépasse une correction triviale.

Une PR n’est pas
Un synonyme de push, un déploiement, ni une preuve automatique de qualité. Elle est un point de contrôle social et technique.

Commit, push, PR, merge : la séquence

1. Commit local
   Instantané cohérent et message expliquant pourquoi.
2. Push
   Publication de la branche sur le serveur Git.
3. PR
   Comparaison, explication, tests automatiques et revue.
4. Merge
   Intégration dans la branche de référence après accord.
5. Tag ou release
   Nom stable donné à la version publiable.
6. Déploiement
   Installation de cette version dans un environnement.

Messages utiles

```
feat(commandes): empêcher la création en double
fix(upload): reprendre après expiration du lien
docs(architecture): consigner le choix du stockage objet
```

Stratégie de branches pragmatique

Pour une petite équipe, une branche principale toujours publiable et des branches courtes suffisent. Une branche longue accumule conflits, dérive et fausses hypothèses. Fusionne souvent derrière un feature flag si la fonctionnalité n’est pas encore exposable. Git Flow peut servir des cycles de versions lourds, mais il n’est pas une vertu universelle.

Conflit Git : un conflit de sens

Un conflit indique que deux histoires modifient la même zone. Ne choisis pas mécaniquement « ours » ou « theirs ». Comprends les deux intentions, reconstruis le résultat attendu, relance les tests puis lis le diff final. L'agent peut aider, mais l'arbitrage appartient au propriétaire du comportement.

Rebase ou merge ?

Rebase rejoue des commits sur une nouvelle base et produit une histoire linéaire ; merge conserve explicitement la réunion. Rebase est pratique avant partage ou sur une branche personnelle. Ne réécris pas sans coordination l'histoire que d'autres utilisent. La politique doit être simple et stable.

Tag, version et release

Un tag nomme un commit. Une version sémantique exprime souvent majeur.mineur.correctif. Une release associe tag, notes, artefacts et parfois approbation. L'important est la traçabilité : depuis un incident, retrouver le code, la configuration, la migration et l'artefact réellement exécutés.

CHAPITRE 7

Tests, revue et preuves

Un test n’est pas une cérémonie. C’est une affirmation exécutable sur un comportement important.

La pyramide pragmatique

Les tests unitaires vérifient vite une règle isolée. Les tests d’intégration vérifient la coopération avec la base, le système de fichiers ou une API. Les tests de bout en bout parcourent un scénario utilisateur réel, plus coûteux et plus fragile. Ajoute des tests contractuels aux frontières critiques.

Que tester en priorité ?

1. Règles métier
   Calculs, transitions, permissions, unicité.
2. Frontières
   Base, stockage, paiement, courriel, API.
3. Échecs coûteux
   Perte, duplication, divulgation, blocage.
4. Régressions
   Chaque bug corrigé reçoit une preuve qui aurait dû le détecter.
5. Parcours vitaux
   Connexion, commande, paiement, export, restauration.

CI : l’atelier automatique

L’intégration continue exécute sur chaque push ou PR une recette propre : installer les dépendances verrouillées, analyser le code, lancer les tests, construire l’artefact. Si la CI échoue, la PR ne doit normalement pas être fusionnée. Le vert prouve uniquement ce qui a été testé.

Relire un diff

Lis le diff comme une décision : chaque fichier sert-il l’objectif ? Une dépendance ou une migration apparaît-elle ? Une erreur est-elle masquée ? Des logs contiennent-ils des données sensibles ? Les tests échoueraient-ils si on retirait le correctif ? Le changement est-il réversible ?

> Preuve minimale avant publication
> Suite automatique verte + test du scénario réel + revue du diff + sauvegarde ou retour arrière approprié + personne clairement responsable de l’observation.

Revue agentique indépendante

Après l’implémentation, donne à une session fraîche l’objectif, le diff et les critères, sans lui demander de défendre la solution. Sa mission : chercher les hypothèses cachées, risques, cas limites et tests manquants. L’agent bâtisseur corrige ensuite ; le décideur arbitre.

Doubles, mocks et faux services

Un fake possède un comportement simplifié mais réel, par exemple une base en mémoire. Un stub renvoie des réponses préparées. Un mock vérifie des interactions attendues. Trop de mocks couplent les tests à l’implémentation et donnent du vert sans prouver le système. Préfère tester les sorties et invariants ; réserve les mocks aux frontières lentes ou difficiles à provoquer.

Tests déterministes

Un test doit produire le même résultat dans les mêmes conditions. Injecte l'horloge, les identifiants aléatoires et les services externes. Isole les données. Attends un événement explicite plutôt qu'un délai arbitraire. Un test intermittent n'est pas un désagrément : il détruit la confiance dans tout le pipeline.

Sécurité comme propriété vérifiable

Teste l'autorisation côté serveur, la validation des entrées, la non-divulgation des erreurs, les limites de taille et de fréquence, la rotation des sessions, et l'impossibilité d'accéder à l'objet d'un autre utilisateur. Les scanners de dépendances complètent cette approche mais ne remplacent pas le modèle de menace.

Matrice risque / niveau de preuve

Une couleur d'interface demande peu de preuve. Une règle de prix exige tests unitaires et scénario métier. Une migration de données exige répétition sur copie, sauvegarde et vérification. Un paiement ou une permission exige tests d'intégration, idempotence, audit et observation. Le coût de vérification doit suivre le coût de l'échec.

CHAPITRE 8

Du local au VPS

Publier n’est pas copier des fichiers au hasard : c’est promouvoir une version identifiée à travers des environnements maîtrisés.

Les environnements

1. Local
   Rapide, isolé, données factices, outils de développement.
2. CI
   Machine propre qui reproduit les vérifications.
3. Staging
   Configuration proche de la production pour essais réalistes.
4. Production
   Service réel, données réelles, accès contrôlé et observation continue.

Le pipeline recommandé

1	Coder sur une branche Tranche limitée et critères explicites.
2	Tester localement Tests ciblés puis suite pertinente.
3	Commit et push Version identifiable sur le dépôt distant.
4	Ouvrir la PR Description, captures, risques, migration, rollback.
5	Laisser la CI prouver Tests, lint, sécurité, construction.
6	Faire relire et merger Accord humain ou politique définie.
7	Construire un artefact Image Docker ou paquet immuable lié au commit.
8	Déployer en staging Migration, smoke tests, parcours critique.
9	Promouvoir en production Même artefact, variables et secrets de production.
10	Observer Santé, erreurs, latence, parcours métier, puis décision de poursuivre ou revenir.

Ce qui vit sur le VPS

Le reverse proxy reçoit HTTPS et route vers l’application. L’application s’exécute comme service ou conteneur. La base persiste sur un volume protégé. Les secrets viennent de l’environnement ou d’un coffre, jamais du dépôt. Les journaux et métriques quittent idéalement la machine ou sont conservés avec rotation.

Migration de base

Une migration est du code versionné qui transforme le schéma. Sauvegarde avant une transformation risquée. Préfère les changements compatibles en plusieurs temps : ajouter, remplir, basculer, puis supprimer plus tard. Une ancienne version doit pouvoir cohabiter pendant le déploiement si le trafic continue.

Smoke test et contrôle de santé

Le health check dit que le processus répond. Le smoke test confirme quelques fonctions vitales : page accessible, connexion base, authentification, création contrôlée, file de tâches. Il ne remplace pas la suite de tests.

> La phrase à exiger
> « La production exécute l’artefact construit depuis le commit X ; la migration Y a réussi ; les smoke tests Z sont verts ; le rollback consiste à… »

Configuration et secrets

Le code reste identique entre staging et production ; la configuration change. Les variables non secrètes peuvent être versionnées par environnement. Les secrets sont stockés dans un gestionnaire ou dans la plateforme avec accès minimal. Valide au démarrage que la configuration requise existe et refuse une valeur dangereuse par défaut.

Artefact immuable

Construire une fois puis promouvoir le même artefact évite que staging et production exécutent des résultats différents. L'image ou le paquet porte l'identifiant du commit et, idéalement, une signature ou somme de contrôle. Modifier manuellement des fichiers sur le VPS détruit cette traçabilité.

Déploiements progressifs

Un rolling deployment remplace les instances progressivement. Blue/green prépare un environnement complet puis bascule le trafic. Canary expose une petite fraction des utilisateurs. Un feature flag sépare mise en production du code et activation fonctionnelle. Le choix dépend du risque, du trafic et de la capacité d'observation.

Une exploitation minimale crédible

1. Entrée sécurisée
   DNS, HTTPS, reverse proxy et renouvellement des certificats.
2. Processus supervisé
   Redémarrage contrôlé, limites et état de santé.
3. Persistance
   Volumes nommés, permissions et sauvegardes hors machine.
4. Journalisation
   Rotation, corrélation et conservation proportionnée.
5. Surveillance
   Disponibilité, saturation, erreurs et métriques métier.
6. Maintenance
   Correctifs système, dépendances et exercice de restauration.

CHAPITRE 9

Optimiser sans abîmer

L’optimisation commence par une mesure, vise un goulot réel et conserve une preuve de comportement.

Les trois coûts

Le temps machine : CPU, mémoire, disque, réseau. Le temps humain : compréhension, modification, diagnostic. Le risque : panne, corruption, sécurité. Un code plus court mais opaque peut économiser des millisecondes et coûter des journées.

La méthode de performance

1. Définir
   Quel parcours est trop lent ? Quel objectif chiffré ?
2. Mesurer
   Profilage, requêtes, traces, percentile p95 plutôt qu’impression.
3. Localiser
   Trouver le goulot dominant.
4. Changer
   Une hypothèse et une modification à la fois.
5. Comparer
   Même charge, avant/après, comportement inchangé.
6. Surveiller
   Vérifier en production et prévenir la régression.

Les gains fréquents

Éviter les requêtes répétées, ajouter un index justifié, paginer, réduire les données transférées, mettre en cache une donnée stable, déplacer un travail long en tâche asynchrone, compresser les images, supprimer une dépendance lourde. Ne cache pas avant d'avoir compris l'invalidation.

Optimiser le code pour l'humain

Noms précis, fonctions courtes autour d'une intention, erreurs explicites, peu d'état global, dépendances injectées, commentaires sur le pourquoi, documentation proche du code. La duplication locale peut être moins coûteuse qu'une abstraction prématurée.

> Signal d'alarme
> Si l'agent propose microservices, cache distribué, file de messages et orchestration sans métriques ni contrainte prouvée, demande le problème exact que chaque élément résout.

Complexité algorithmique

La notation O décrit la croissance du coût quand la taille augmente. O(1) reste approximativement constant, O(log n) croît lentement, O(n) suit la taille, $O(n^2)$ devient vite coûteux. Elle ne remplace pas la mesure : un petit n, les accès disque, le réseau et les constantes peuvent dominer. Elle aide surtout à repérer une conception qui ne passera pas à l'échelle.

Base de données : regarder les plans

Une requête lente doit être expliquée par son plan d’exécution : index utilisé, lignes estimées et lues, tris, jointures. Un index accélère certaines lectures mais coûte espace et écritures. Le problème N+1 apparaît lorsqu’une liste déclenche une requête supplémentaire par élément. Mesure avec un jeu de données représentatif.

Cache : une copie avec une date de péremption

Avant d’ajouter un cache, nomme la source de vérité, la clé, la durée, la stratégie d’invalidation, la taille et le comportement si le cache tombe. Une donnée incorrecte très rapide reste incorrecte. Le cache local est simple mais divergent entre instances ; le cache partagé ajoute un service à exploiter.

Budget de performance

Fixe des objectifs par parcours : p95 sous 400 ms, fichier initial sous 250 ko, traitement asynchrone démarré sous 2 s. Le percentile p95 signifie que 95 % des observations sont plus rapides. Une moyenne peut masquer une minorité d’utilisateurs très mal servis.

CHAPITRE 10

Incidents, sauvegardes et retour arrière

Un système professionnel ne promet pas l’absence de panne. Il réduit l’impact, détecte vite et sait restaurer.

Rollback et roll-forward

Le rollback remet l’ancienne version. Le roll-forward publie un correctif. Une migration destructive peut rendre le rollback impossible ; d’où les migrations compatibles et les sauvegardes testées. Décide avant le déploiement quel signal déclenche l’arrêt.

Sauvegarder n’est pas restaurer

Une sauvegarde utile est automatique, chiffrée, datée, conservée hors du VPS et régulièrement restaurée sur un environnement de test. Définis le RPO - quantité maximale de données acceptables à perdre - et le RTO - temps maximal acceptable pour restaurer le service.

La procédure d’incident

1. Détecter
   Alerte ou signal utilisateur qualifié.
2. Stabiliser
   Arrêter l’aggravation : désactiver, isoler, revenir.
3. Communiquer
   Impact, responsable, prochaine mise à jour.
4. Diagnostiquer
   Chronologie, logs, métriques, changements récents.
5. Rétablir
   Retour arrière, restauration ou correctif vérifié.
6. Apprendre
   Post-mortem sans blâme, actions datées et test de non-régression.

Observabilité

Les logs racontent des événements. Les métriques quantifient dans le temps. Les traces suivent une requête à travers les composants. Ajoute des identifiants de corrélation, des niveaux de logs et des alertes liées à l’expérience réelle, pas seulement au CPU.

Avant une opération risquée
Sauvegarde vérifiée, version actuelle identifiée, commande exacte, cible exacte, temps estimé, critère de succès, personne disponible et procédure de retour.

Post-mortem utile

Établis une chronologie factuelle, l’impact, les mécanismes techniques et organisationnels, ce qui a limité l’incident et ce qui a retardé la réponse. Une cause racine unique est souvent une fiction : plusieurs barrières ont manqué. Les actions doivent avoir un propriétaire, une échéance et un test de réussite.

SLO et budget d’erreur

Un indicateur SLI mesure un service, par exemple le taux de requêtes valides. Un objectif SLO fixe la cible, par exemple 99,9 % sur trente jours. Le budget d’erreur est la part d’échec tolérée ; lorsqu’il est consommé, on ralentit les changements risqués et travaille la fiabilité. Cette logique transforme « stable » en décision quantifiée.

CHAPITRE 11

La méthode ORCHESTRE

Une méthode courte à appliquer à chaque fonctionnalité, correction ou publication.

O	Observer Le système réel, ses contraintes et les preuves.
R	Résultat Formuler la valeur et les critères d’acceptation.
C	Cartographier Données, flux, frontières, risques et propriétaires.
H	Hypothèse Choisir la solution minimale et ses conséquences.
E	Expérimenter Créer une preuve qui échoue, puis implémenter.
S	Sécuriser Tests, droits, erreurs, sauvegarde, cas limites.
T	Tracer Commit, ADR, documentation et journal de version.
R	Relire Diff et revue indépendante.

E Exposer
Staging, production progressive, observation, rollback.

Le point d’arrêt décisionnel

Après Cartographier et Hypothèse, arrête l’agent si la solution crée une nouvelle dépendance majeure, modifie les données, touche à l’authentification, engage un fournisseur ou change l’architecture. Tu arbitres alors avec un ADR court. Le reste peut suivre automatiquement dans le périmètre autorisé.

Le compte rendu idéal de l’agent

Livraison
Résultat obtenu ; fichiers et décisions ; tests exécutés avec résultats ; éléments non testés ; risques restants ; migration éventuelle ; commandes de publication ; méthode de retour arrière.

Ta responsabilité

Tu n’as pas à vérifier chaque caractère. Tu dois vérifier l’alignement entre problème, décision, preuve et risque. Quand l’un manque, le travail n’est pas prêt. L’agent est responsable de montrer son raisonnement opératoire ; toi, de décider ce qui mérite confiance.

CHAPITRE 12

Cas pratique : créer une fonctionnalité

Nous allons ajouter un dépôt de fichier client avec reprise, contrôle de taille et confirmation, depuis la demande brute jusqu’à la production.

Le besoin brut

« Je veux que le client puisse déposer son PDF de fabrication, jusqu’à 500 Mo, sans recommencer si le réseau coupe. L’atelier doit voir le fichier dans la commande. » Cette phrase contient valeur, mais pas encore contrat, sécurité, états ni politique d’échec.

Étape 1 - cadrer avec l’agent

```
Utilisateur : Analyse le dépôt. Ne modifie rien. Cartographie le parcours de commande, le stockage actuel, les limites d’upload, les règles d’accès et les tests. Propose les inconnues qui empêchent de définir la fonctionnalité.

Agent attendu : état Git, fichiers et composants concernés, flux actuel, limites observées, risques et questions métier.
```

L’agent découvre par exemple une interface web, une API, PostgreSQL pour les métadonnées et un stockage objet. Il constate que l’API accepte aujourd’hui des fichiers de 20 Mo en transitant par le serveur. Le problème n’est donc pas seulement la taille du champ : le serveur deviendrait un goulot et une coupure ferait perdre tout le transfert.

Étape 2 - transformer le besoin en contrat

1. Acteur
   Client authentifié ayant accès à la commande.
2. Précondition
   Commande ouverte et autorisée à recevoir un fichier.
3. Entrée
   PDF, 500 Mo maximum, nom normalisé.
4. Parcours
   Créer une session, transférer par morceaux, finaliser, analyser.
5. États
   préparé, en cours, reçu, validé, rejeté, expiré.
6. Sortie
   Pièce jointe visible avec taille, date et statut.
7. Échecs
   Coupure, doublon, type incorrect, session expirée, quota dépassé.
8. Invariant
   Un objet n’est exploitable par l’atelier qu’après finalisation et validation.

Étape 3 - choisir l’architecture minimale

Le navigateur demande à l’API une session d’upload. L’API vérifie droits et quota, crée un enregistrement et retourne une autorisation temporaire limitée à une clé. Le navigateur envoie directement les morceaux au stockage objet. Il appelle ensuite finaliser. L’API vérifie taille et présence, marque reçu, puis une tâche analyse le fichier avant de le rendre disponible.

Pourquoi cette architecture ?
Le gros flux ne traverse pas l'application ; le secret du stockage n'est jamais donné ; l'autorisation expire ; l'état métier reste en base ; la reprise utilise les parties déjà reçues.

Étape 4 - enregistrer les décisions

ADR : upload direct multipart vers stockage objet. Alternatives rejetées : transit par API, trop coûteux et fragile ; serveur FTP séparé, identité et expérience dupliquées. Conséquences : gestion des uploads abandonnés, CORS, nettoyage périodique, validation asynchrone et tests contractuels du fournisseur.

Étape 5 - découper en tranches verticales

1. Tranche A
   Créer une session et afficher un fichier fictif reçu.
2. Tranche B
   Envoyer réellement un petit PDF et finaliser.
3. Tranche C
   Ajouter multipart et reprise après coupure.
4. Tranche D
   Validation, rejet et visibilité atelier.
5. Tranche E
   Nettoyage, métriques, quotas et durcissement.

Chaque tranche peut être démontrée. On évite une branche de trois semaines qui mélange interface, stockage, sécurité et traitement.

Étape 6 - mission d’implémentation

```
Objectif : implémente la tranche A uniquement.
Contraintes : conserve l’adaptateur de stockage existant ; aucune nouvelle dépendance sans justification ; pas de migration destructive.
Acceptation : un client autorisé crée une session ; un autre reçoit 403 ; taille supérieure reçoit 422 ; deux requêtes avec la même clé d’idempotence retournent la même session.
Méthode : ajoute d’abord les tests ; montre le plan de migration ; exécute tests ciblés puis suite du module ; relis le diff.
Arrêt : demande avant tout changement d’authentification ou de schéma non additif.
```

Étape 7 - preuve rouge puis verte

L’agent écrit un test d’intégration qui appelle l’endpoint inexistant : 404 au lieu de 201. Cette première erreur confirme que le test observe le manque. Il ajoute ensuite le cas d’usage, la table additive et l’adaptateur. Les tests passent. Pour vérifier que le test n’est pas décoratif, la revue peut désactiver temporairement la contrainte d’idempotence et constater l’échec.

Étape 8 - revue du diff

1. Intention
   Chaque fichier sert la création de session.
2. Modèle
   Contrainte unique sur organisation et clé d’idempotence.
3. Sécurité
   Autorisation calculée côté serveur ; clé de stockage non choisie par le client.
4. Erreurs
   403, 422 et conflit sont structurés et non bavards.
5. Observabilité
   Identifiant d’upload dans logs, sans URL signée.
6. Migration
   Ajout compatible et retour possible tant qu’aucune ancienne colonne n’est supprimée.

Étape 9 - PR et CI

La branche est poussée. La PR explique le besoin, le flux, l’ADR, les captures, les commandes de test, la migration et le risque. La CI reconstruit sur une machine propre. Une revue indépendante tente : double clic, session expirée, accès croisé entre clients, fichier annoncé différent du fichier reçu, fournisseur indisponible.

Étape 10 - staging

Déploie le même artefact candidat. Utilise un bucket de staging et des comptes factices. Simule une coupure après plusieurs parties, recharge la page et reprends. Vérifie que les uploads abandonnés expirent, que les métriques bougent et que les journaux permettent de suivre un identifiant sans exposer de secret.

Étape 11 - production progressive

La migration additive part d'abord. Puis l'artefact est déployé avec la fonctionnalité désactivée. On active pour un compte interne, puis quelques clients. On observe taux de succès, durée p95, abandons, erreurs fournisseur et espace orphelin. Le rollback désactive le flag et revient à l'artefact précédent ; la table additive peut rester jusqu'à une migration ultérieure.

Le handoff final de l'agent

> Compte rendu attendu
> Version et commit ; architecture retenue ; migrations ; tests exécutés et résultats ; scénario staging ; métriques ; limites connues ; procédure d'activation ; rollback ; travail explicitement reporté.

Ce que l'utilisateur a appris

La fonctionnalité n'est pas « un bouton upload ». C'est une machine à états traversant navigateur, autorisation, stockage, base et traitement. La bonne abstraction rend les pannes visibles et récupérables. Le rôle de l'orchestrateur est de faire émerger ce système avant qu'il ne soit caché dans du code.

CHAPITRE 13

Cas pratique : diagnostiquer un bug

Nous allons traiter un doublon de commande intermittent sans sauter directement sur le premier correctif plausible.

Le signal

Le support rapporte : « Certains clients voient deux commandes après avoir cliqué sur Valider. » Le défaut est rare et impossible à reproduire à volonté. Modifier le bouton immédiatement serait tentant, mais une protection d’interface ne garantit pas l’intégrité du serveur.

Étape 1 - figer les faits

```
Mission : diagnostique sans modifier. Cherche les commandes dupliquées, événements, logs et changements récents. Établis une chronologie avec identifiants de requête et de paiement. Distingue faits, hypothèses et informations manquantes. Ne consulte que les données autorisées et masque les informations personnelles.
```

On observe deux créations séparées de 180 ms, même utilisateur, même panier, deux identifiants de requête. Le premier appel a répondu lentement. Aucun paiement double. Un déploiement récent a changé le délai du client, mais pas le service de commande.

Étape 2 - construire l’espace des hypothèses

1. Double interaction
   Double clic ou touche Entrée et clic.
2. Nouvel essai automatique
   Client, proxy ou bibliothèque répète après délai.
3. Webhook répété
   Un événement externe recrée la commande.
4. Course serveur
   Deux workers passent la vérification avant l’écriture.
5. Mauvais affichage
   Une commande jointe deux fois sans duplication réelle.
6. Ancienne version
   Instances de versions différentes se comportent autrement.

Étape 3 - classer par preuve

La base contient bien deux lignes : ce n’est pas un affichage. Les traces montrent deux requêtes du navigateur : le webhook est innocent. Les requêtes passent simultanément dans « aucune commande existante », puis insèrent. L’hypothèse de course est forte ; le double clic explique le déclencheur, pas la vulnérabilité.

Étape 4 - reproduire

L’agent ajoute un test d’intégration concurrent : deux requêtes avec la même intention sont libérées ensemble par une barrière. Sans correction, deux lignes apparaissent. Répéter le test cent fois n’est pas une solution déterministe ; la synchronisation du test force précisément la fenêtre critique.

Principe
Un bon test de régression reproduit le mécanisme, pas seulement le symptôme. Sinon il peut rester vert tout en laissant la course intacte.

Étape 5 - choisir la barrière fiable

Le bouton est désactivé pour améliorer l'expérience, mais la garantie repose sur une clé d'idempotence issue de l'intention client et une contrainte unique en base. Le service tente l'insertion dans une transaction ; en cas de conflit, il retourne la commande existante. Cette défense fonctionne même avec plusieurs instances.

Étape 6 - mission de correction

```
Objectif : empêcher une création multiple pour la même intention.
Preuve : le test concurrent doit échouer avant et passer après ; les intentions différentes restent indépendantes.
Contraintes : ne fusionne pas des commandes historiques ; migration additive ; préserve la compatibilité avec les clients sans clé pendant une période définie.
Livraison : diff minimal, plan de remplissage, métrique de conflits et rollback.
```

Étape 7 - examiner les angles morts

1. Portée de la clé
   Par utilisateur, organisation ou panier ?
2. Durée
   Une clé peut-elle être réutilisée demain ?
3. Réponse
   Que renvoyer si le premier traitement est encore en cours ?
4. Échec partiel
   La clé reste-t-elle bloquée après un échec récupérable ?
5. Anciennes données
   Existe-t-il déjà des doublons empêchant l’index unique ?
6. Effets secondaires
   Courriel ou paiement sont-ils eux-mêmes idempotents ?

Étape 8 - migration sûre

Ajouter la colonne nullable ; déployer le code capable de lire ancien et nouveau ; renseigner progressivement si nécessaire ; mesurer les collisions ; créer l’index unique lorsque les données sont propres ; rendre obligatoire plus tard. Le principe expand/contract évite de casser l’ancienne version pendant le rolling deployment.

Étape 9 - vérification

Tests unitaires de la clé, test concurrent avec vraie base, test API, suite du module, analyse de migration sur copie représentative, et parcours manuel double clic. La revue retire temporairement l’index : le test concurrent doit échouer. On confirme que les courriels ne partent qu’après commit.

Étape 10 - publication et observation

Déploiement avec métriques : nombre de clés répétées, conflits uniques, créations et erreurs. Un pic peut signaler une boucle client. Le seuil d’arrêt porte sur erreurs et latence. Le rollback applicatif reste possible tant que le schéma est additif ; retirer précipitamment la colonne n’apporte rien.

Étape 11 - post-mortem

Cause technique : vérification puis insertion non atomiques. Déclencheur : nouvel essai client après délai. Barrières manquantes : idempotence, contrainte unique et test concurrent. Actions : convention d’idempotence pour toutes les créations critiques, identifiant de corrélation, test générique et documentation du contrat.

Dialogue de clôture

```
Utilisateur : Qu’est-ce qui prouve que la cause est corrigée, et non masquée ?
Agent : le test force deux transactions concurrentes ; sans contrainte, deux insertions réussissent ; avec la migration et le traitement du conflit, une seule ligne existe et les deux réponses portent le même identifiant. La métrique de conflit confirmera en production que les répétitions sont absorbées.
```

Ce que l’utilisateur a appris

Déboguer est une enquête causale. On part d’un signal, on conserve les faits, on énumère des hypothèses réfutables, on force une reproduction, on corrige au niveau de garantie approprié et on observe. L’agent accélère la recherche ; le protocole empêche la première intuition de devenir une vérité.

CHAPITRE 14

Fiches réflexes et glossaire

Les pages à ouvrir quand tu pilotes une session, une PR ou une mise en production.

Fiche 1 - Lancer une tâche

1	Donner le contexte But du produit, utilisateurs et zone concernée.
2	Dire le résultat Comportement observable, pas technologie imposée.
3	Fixer les limites Données, sécurité, fichiers, opérations interdites.
4	Définir les preuves Critères, tests, capture ou démonstration.
5	Exiger le retour Diff, décisions, risques et suite.

Fiche 2 - Accepter une PR

```
□ Le besoin est expliqué. □ Le diff est limité. □ La CI est verte. □ Les tests couvrent le risque. □ Les erreurs sont visibles. □ Aucun secret. □ Migration compatible. □ Documentation à jour. □ Rollback crédible.
```

Fiche 3 - Autoriser la production

```
□ Commit/tag identifié. □ Sauvegarde appropriée. □ Artefact immuable. □ Variables et secrets présents. □ Migration répétée en staging. □ Smoke tests définis. □ Observateur nommé. □ Seuil d’arrêt et rollback prêts.
```

Glossaire essentiel

API — Contrat permettant à deux logiciels d’échanger.
Artefact — Résultat construit et déployable lié à une version.
Branche — Pointeur vers une ligne de commits.
Build — Transformation du code source en résultat exécutable.
CI/CD — Vérification continue / livraison ou déploiement automatisé.
Commit — Instantané versionné accompagné d’un message.
Conteneur — Paquet isolé regroupant application et environnement d’exécution.
Déploiement — Mise en service d’une version dans un environnement.
Diff — Comparaison précise entre deux états du code.
Endpoint — Adresse et opération exposées par une API.
Framework — Cadre logiciel structurant l’application.
Idempotence — Propriété d’une opération répétable sans effet supplémentaire.
Lint — Analyse automatique de conventions et erreurs probables.
Merge — Réunion de deux lignes d’histoire Git.
Migration — Transformation versionnée du schéma ou des données.
PR — Proposition documentée d’intégrer une branche.
Refactorisation — Amélioration interne sans changer le comportement.
Régression — Fonctionnement ancien cassé par un changement.
Rollback — Retour à une version antérieure.
Secret — Information sensible nécessaire à l’exécution.
Staging — Environnement de répétition proche de la production.
VPS — Serveur virtuel administré sur lequel tournent les services.
Webhook — Appel automatique envoyé lors d’un événement.

Dernière page

La simplicité est une discipline

Un logiciel simple n’est pas un logiciel naïf. C’est un système dont les responsabilités sont visibles, les décisions tracées, les risques proportionnés et les changements vérifiables. L’agent peut accélérer chaque geste. La méthode conserve la direction.

À garder en tête
Une intention par changement. Une preuve par risque. Une trace par décision. Un retour possible par publication.

Fin de la première édition. Ce manuel est conçu pour être annoté, challengé et transformé en méthode vivante.